

ASSIGNMENT 3

Name: Piyush
UID: 22BCS15782

Section: IOT_608
Group: B

Solution 1:

```
class Solution {
public:
    void printList(Node *head) {
        Node *temp = head;
        while (temp != nullptr) {
            cout << temp->data << " "; // Print data with space
            temp = temp->next;
        }
    }
};
```

The screenshot displays a coding platform interface. On the left, the 'Output Window' shows 'Compilation Results' for 'Y.O.G.I. (AI Bot)'. It indicates 'Problem Solved Successfully' with 1112 test cases passed out of 1112, an accuracy of 25%, 1 point scored out of 1, and a time taken of 0.07. Below this, 'Solve Next' buttons are visible: 'Count Linked List Nodes', 'Delete Alternate Nodes', and 'Insert in Middle of Linked List'. On the right, the code editor shows the C++ code for the 'printList' function, which iterates through a linked list and prints each node's data with a space separator. The code is enclosed in a class 'Solution'.

Solution 2:

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;

        while (current != nullptr && current->next != nullptr) {
            if (current->val == current->next->val) {
```

```

        ListNode* temp = current->next;
        current->next = current->next->next; // Skip the duplicate node
        delete temp; // Free memory
    } else {
        current = current->next; // Move to the next node
    }
}
return head;
}
};

```

The screenshot shows a submission interface for a C++ problem. The submission is marked as 'Accepted' with 168/168 testcases passed. The user 'young_lord_07' submitted it on Mar 07, 2025 at 12:53. The runtime is 0 ms, beating 100.00% of other submissions. The memory usage is 16.24 MB, beating 35.06% of other submissions. A bar chart shows the runtime distribution, with a single bar at 0 ms. The code editor on the right shows the implementation of the 'deleteDuplicates' function, which uses a two-pointer approach to remove duplicates from a linked list. The test result section shows 'Accepted' with a runtime of 0 ms and an input of 'head = [1,1,2]'.

Solution 3:

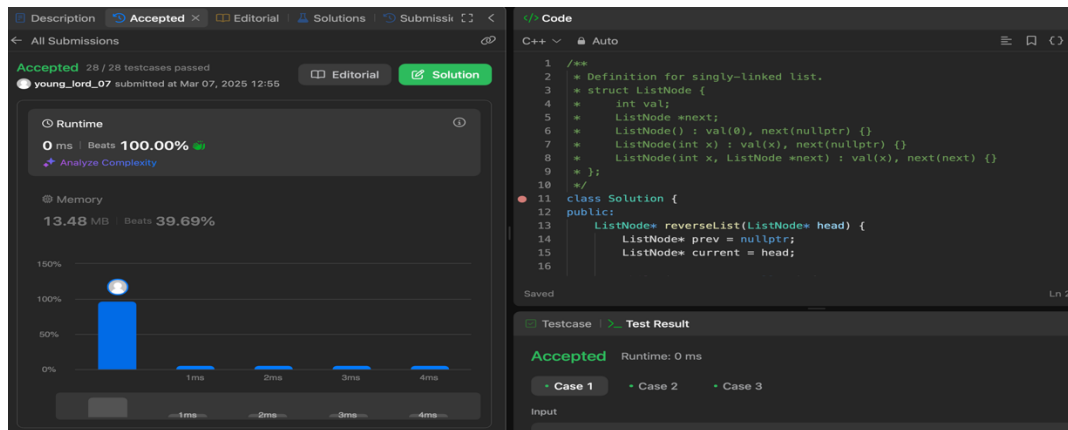
```

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* current = head;

        while (current != nullptr) {
            ListNode* nextNode = current->next; // Store next node
            current->next = prev; // Reverse the link
            prev = current; // Move prev forward
            current = nextNode; // Move current forward
        }

        return prev; // New head of reversed list
    }
};

```



Solution 4:

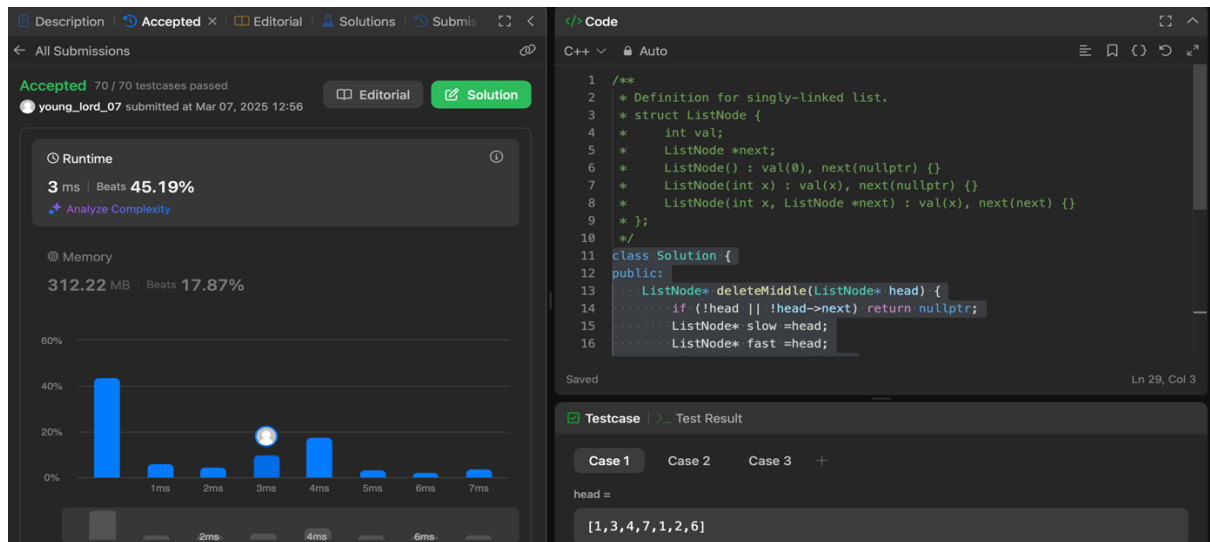
```

class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;

        while(fast && fast -> next){
            prev = slow;
            slow = slow -> next;
            fast = fast -> next -> next;
        }

        prev -> next = slow -> next;
        delete slow;
        return head;
    }
};

```



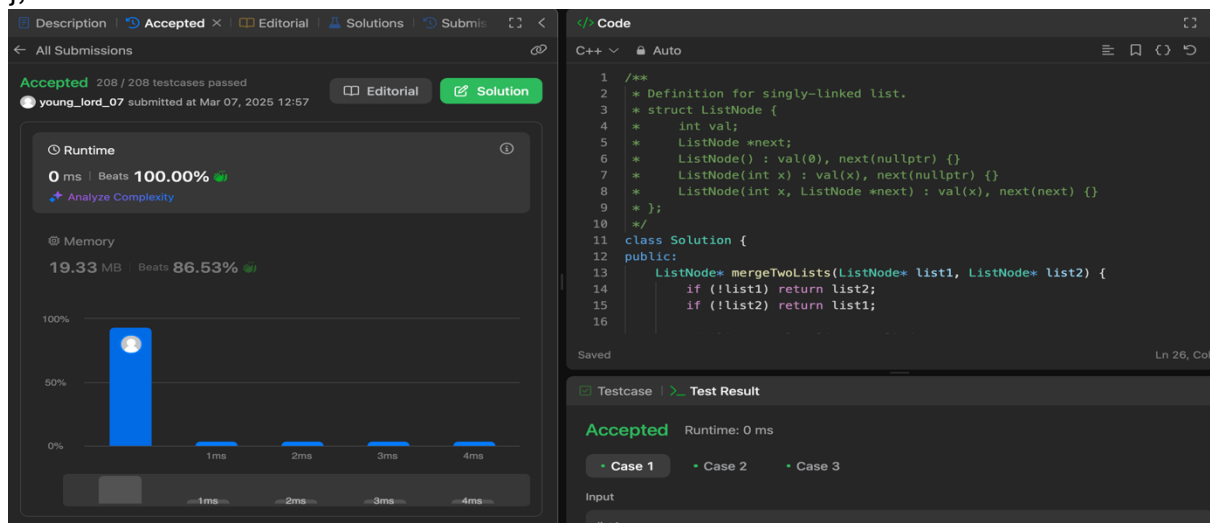
Solution 5:

```

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;

        if (list1->val < list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};

```



Solution 6:

```

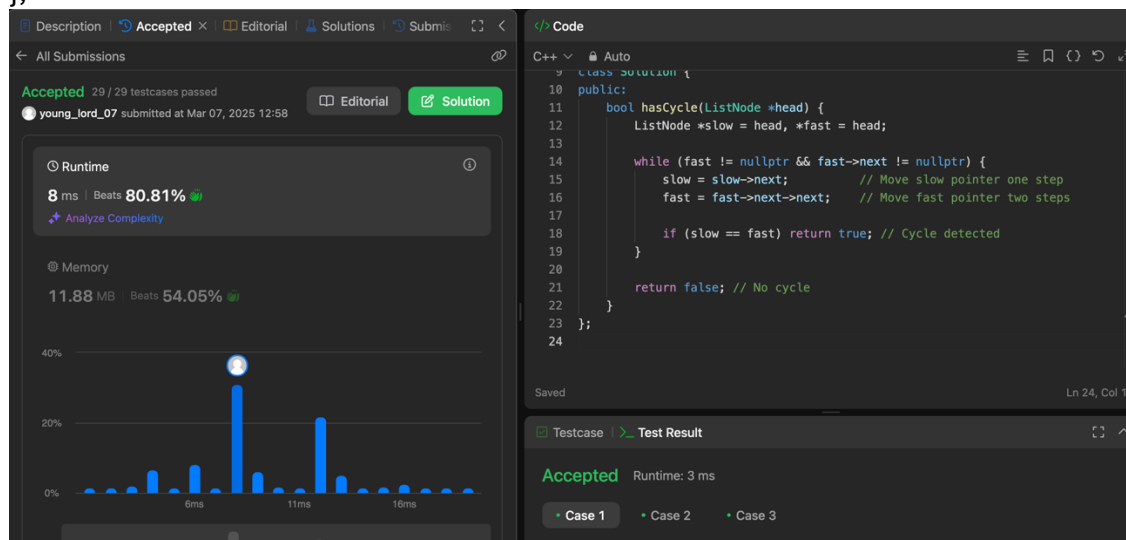
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;

        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;    // Move slow pointer one step
            fast = fast->next->next; // Move fast pointer two steps

            if (slow == fast) return true; // Cycle detected
        }

        return false; // No cycle
    }
};

```



Solution 7:

```

class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head; // Handle edge cases

        // Step 1: Compute the length of the list
        int length = 1; // Start from 1 since we count from the first node
        ListNode* tail = head;
        while (tail->next) {
            tail = tail->next;
            length++;
        }
    }
};

```

// Step 2: Compute the effective rotation (as rotating `length` times gives the same list)

```
k = k % length;
```

```
if (k == 0) return head; // If `k` is a multiple of `length`, no change is needed
```

// Step 3: Find the new tail (length - k - 1) and new head (length - k)

```
ListNode* newTail = head;
```

```
for (int i = 0; i < length - k - 1; i++) {
```

```
    newTail = newTail->next;
```

```
}
```

// Step 4: Rotate the list

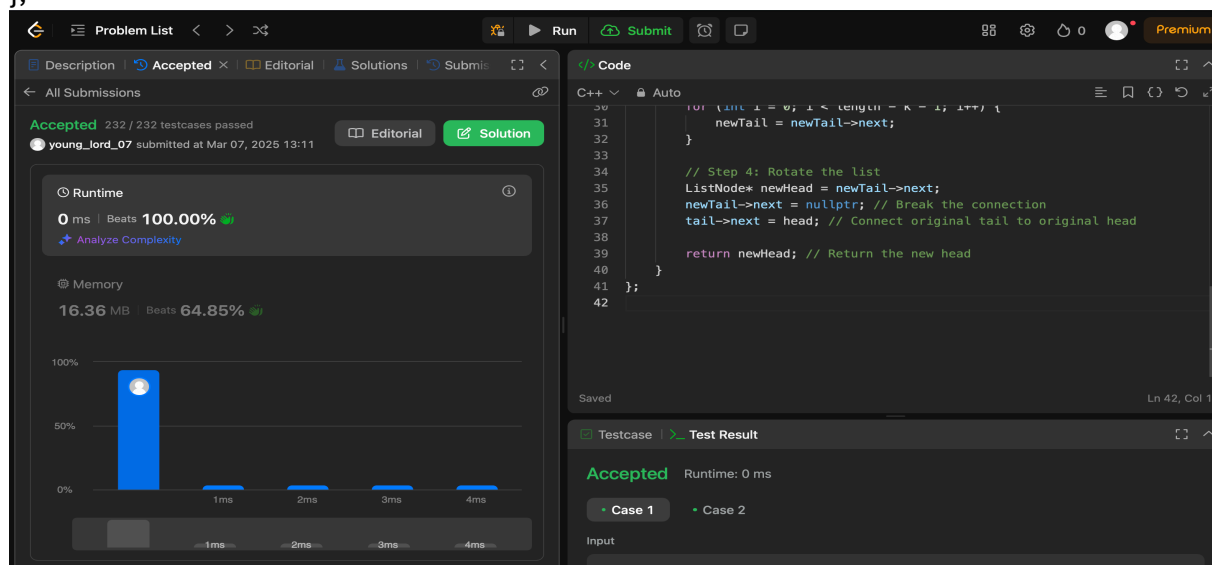
```
ListNode* newHead = newTail->next;
```

```
newTail->next = nullptr; // Break the connection
```

```
tail->next = head; // Connect original tail to original head
```

```
return newHead; // Return the new head
```

```
}  
};
```



```

}

// Function to merge two sorted linked lists
ListNode* merge(ListNode* left, ListNode* right) {
    ListNode dummy(0);
    ListNode* tail = &dummy;

    while (left && right) {
        if (left->val < right->val) {
            tail->next = left;
            left = left->next;
        } else {
            tail->next = right;
            right = right->next;
        }
        tail = tail->next;
    }

    // Append remaining nodes
    tail->next = left ? left : right;

    return dummy.next;
}

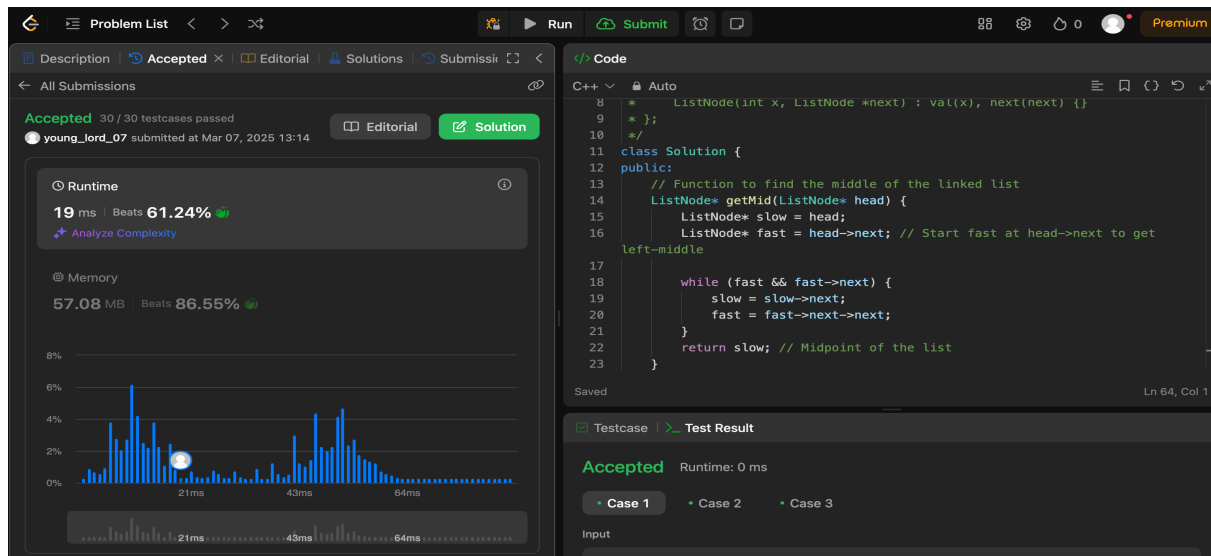
// Main function to sort the list using Merge Sort
ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head; // Base case (1 or 0 nodes)

    // Step 1: Split list into two halves
    ListNode* mid = getMid(head);
    ListNode* rightHead = mid->next;
    mid->next = nullptr; // Break the list into two halves

    // Step 2: Recursively sort both halves
    ListNode* left = sortList(head);
    ListNode* right = sortList(rightHead);

    // Step 3: Merge two sorted halves
    return merge(left, right);
}
};

```



Solution 9:

```
#include <queue>
```

```

class Solution {
public:
    struct Compare {
        bool operator()(ListNode* a, ListNode* b) {
            return a->val > b->val; // Min-heap based on value
        }
    };
};

```

```

ListNode* mergeKLists(vector<ListNode*>& lists) {
    priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;

```

```

    // Push all non-null list heads into the heap
    for (ListNode* list : lists) {
        if (list) minHeap.push(list);
    }

```

```

    ListNode dummy(0);
    ListNode* tail = &dummy;

```

```

    while (!minHeap.empty()) {
        ListNode* node = minHeap.top();
        minHeap.pop();
        tail->next = node;
        tail = tail->next;

        if (node->next) minHeap.push(node->next); // Push next node of extracted list
    }

```



```
    return dummy.next;
}
};
```

Problem List < > < > Run Submit

Description Accepted Editorial Solutions Submissi

All Submissions

Accepted 134 / 134 testcases passed

young_lord_07 submitted at Mar 07, 2025 13:13

Editorial Solution

Runtime

3 ms | Beats 64.89%

Analyze Complexity

Memory

18.41 MB | Beats 66.14%

75% 50% 25% 0%

1ms 50ms 100ms 150ms

1ms 50ms 100ms 150ms

Code

C++ v Auto

```
1 /**
2  * Definition for singly-linked list.
3  * struct ListNode {
4  *     int val;
5  *     ListNode *next;
6  *     ListNode() : val(0), next(nullptr) {}
7  *     ListNode(int x) : val(x), next(nullptr) {}
8  *     ListNode(int x, ListNode *next) : val(x), next(next) {}
9  * };
10 */
11 #include <queue>
12
13 class Solution {
14 public:
15     struct Compare {
16         bool operator()(ListNode* a, ListNode* b) {
```

Saved Ln 11, Col 1

Testcase > Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input